Aggressive Slack Recycling via Transparent Pipelines

Gokul Subramanian Ravi
University of Wisconsin - Madison
gravi@wisc.edu

Mikko H. Lipasti
University of Wisconsin - Madison
mikko@engr.wisc.edu

ABSTRACT

In order to operate reliably and produce expected outputs, modern architectures set timing margins conservatively at design time to support extreme variations in workload and environment. Unfortunately, the conservative guard bands set to achieve this reliability create *clock cycle slack* and are detrimental to performance and energy efficiency. To combat this, we propose *Aggressive Slack Recycling via Transparent Pipelines*. Our proposal performs timing speculation while allowing data to flow asynchronously via transparent latches, between synchronous boundaries. This allows timing speculation to cater to the average slack across asynchronous operations rather than the slack of the most critical operation - maximizing slack conservation and timing speculation efficiency.

We design a slack tracking mechanism which runs in parallel with the transparent data path to estimate the accumulated slack across operation sequences. The mechanism then appropriately clocks synchronous boundaries early to minimize wasted slack and maximize clock cycle savings. We implement our proposal on a spatial fabric and achieves absolute speedups up to 20% and relative improvements (vs. competing mechanisms) of up to 75%.

CCS CONCEPTS

• **Computer systems organization** → **Serial architectures**;

KEYWORDS

Timing Slack, Transparent Pipeline, Variation

ACM Reference Format:

Gokul Subramanian Ravi and Mikko H. Lipasti. 2018. Aggressive Slack Recycling via Transparent Pipelines. In *ISLPED '18: ISLPED '18: International Symposium on Low Power Electronics and Design, July 23–25, 2018, Seattle, WA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3218603.3218623

1 INTRODUCTION

Modern processing architectures are designed to be reliable. They are designed to operate correctly and efficiently on diverse workloads across varying environmental conditions. To achieve this, the work performed by any functional unit (FU) in a synchronous design should be completed within its clock period, every clock cycle. Thus, conservative timing guard bands are employed to handle wide environmental (PVT) variations as well as all legitimate workload characteristics that might activate the critical path in any FU. In the common non-critical cases, this creates clock cycle *slack* - the fraction of the clock cycle performing no useful work. Under typical conditions and workload characteristics, each clock cycle produces slack averaging more than 25% of the clock period and sometimes even as much as 40% [3]. Performance and/or energy efficiency are thus sacrificed for reliability. Moreover, scaling to lower technology nodes creates an increasing gap between worst-case and nominal circuit delays, requiring even larger guard bands [6].

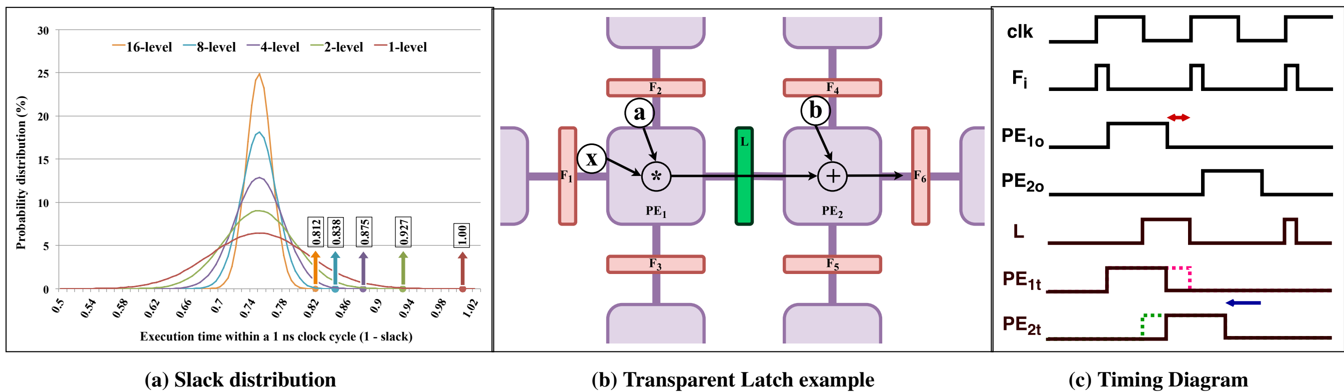
Timing Speculation (TS) is a state-of-the-art mechanism, which cuts into traditional timing guard bands, providing better execution efficiency at the risk of timing violations. When coupled with error detection and recovery, it presents a functionally correct, efficient, processor design. Its prior implementations in the synchronous domain have focused on adaptive variation of the operating points (F,V) by tracking the frequency of timing errors occurrences [1] or by estimating impact of PVT variations on slack [3, 7] and so on.

Prior synchronous TS solutions suffer two fundamental constraints. First, they are bounded by the possibility of timing errors from *every* computation, in *every* synchronous FU or operation stage, and on *every* clock cycle. Second, the dynamic mechanisms among these are implemented by varying frequency/voltage over time and thus, can only be *reconfigured* at a reasonably coarse granularity of time (at best, over epochs of 1000s of cycles). Thus, ensuring no (or minimal) timing errors over the entire epoch forces these operating points to be set rather conservatively, constrained by timing requirements of each operation in the entire epoch. Otherwise, they run the risk of increased timing violations. While recovery mechanisms [1] maintain reliable operation in the face of timing errors, they impose significant penalties on performance and energy efficiency.

On the other hand, purely asynchronous solutions are inherently suited to slack conservation [8]. Varying execution times among operations which could cause timing errors in an aggressive synchronous TS design, can be avoided by allowing such varied delays to be balanced within the entire asynchronous execution window. But pure asynchronous solutions suffer from other functional complexities resulting in low throughput and/or high overhead implementation costs, making them a less popular solution.

To leverage the benefits of asynchronous solutions within the synchronous (pipelined) computing realm, we propose **Aggressive Slack Recycling via Transparent Pipelines**: (1) Simple "asynchronous" execution engines are integrated seamlessly into synchronous pipelines. (2) These engines are implemented as transparent pipelines with synchronous control - resulting in relatively low design complexity. (3) Multiple asynchronously executable operations, bounded by synchronous boundaries, are grouped together into a transparent multi-cycle execution flow. This allows the timing speculation mechanism to cater to the average slack across these grouped executions rather than the most critical operation itself - allowing more aggressive timing speculation. (4) Finally, benefits from timing speculation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISLPED '18, July 23–25, 2018, Seattle, WA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5704-3/18/07...$15.00
https://doi.org/10.1145/3218603.3218623



(a) Slack distribution (b) Transparent Latch example (c) Timing Diagram

Figure 1: Motivating asynchronous timing speculation

are obtained by clocking synchronous boundaries (to the sequences) early, rather than increasing frequency or decreasing voltage, via the use of a reliable slack estimation mechanism.

2 BACKGROUND

PVT Variations: As chip designers attempt to reduce supply voltage to meet power targets, parameter variations are a serious problem. Environment induced variations which affect the functioning of a processor fall into three categories: process, voltage and temperature. Process variations are caused due to wafer characteristics, doping fluctuations etc., leading to potentially large variations in device attributes [3]. They are broken down into Die-to-die (D2D) and Within-Die (WID) variation. WID consists of a systematic component characterized by spatial correlation and a random component with no correlation characteristics [11]. Random variability increases sharply as supply voltage scales down: scaling from 1.0V to 0.3V increases variability by 6x, making this a significant component at NTV [8].

Supply voltage and on-chip temperature also vary with workload and environment. Voltage variations result in current fluctuations on the order of 10s to 100s of cycles [3] and can also exacerbate thermal hot spots. Thermal variations cause changes to leakage current and restrict permissible voltage and TDP in the chip's environment.

Data-dependent Variations: More often than not, a circuit finishes a computation before the worst-case delay elapses since the critical paths might be inactive. For example, the delay of an adder is dependent on the length of its carry propagation. For a double-precision floating point adder, 99.99% of operations settle at the correct output in 87% of the critical adder delay. Moreover, functional units like ALUs perform logical operations consuming less latency (roughly 50%) in comparison to arithmetic operations.

Thus, traditional processors fix operating points conservatively so that even the most critical computations and variations do not violate timing. To improve clock-period utilization and deliver efficient execution, the potential for timing speculation is intuitive and well established.

3 ASYNCHRONOUS TIMING SPECULATION

3.1 Motivation from statistical theory

A popular technique to adaptively control timing guard-bands is Razor [1] which tunes the supply voltage by monitoring the error rate during operation. The commonality among Razor and most TS approaches is that, they all *focus on reducing slack on a per operation basis* and are *constrained by the possibility that timing errors might be caused by every operation, in every unit and on every cycle.* These techniques are tuned to be relatively conservative - having to cater to the most critical operations or execution stages, to prevent mispeculation, or suffer the risk of increasing possibilities of timing errors. The following analysis describes the potential for more aggressive timing speculation and motivates our primary proposal.

Fig.1a shows different slack distributions. Consider the flattest distribution, the *1-level* curve in red. This represents independent and identically distributed logic delay within an FU - averaging at about 75% of the clock cycle but with a reasonably large variance. This curve corresponds to the logic delay experienced on every clock cycle, by a standard design wherein every operation executes synchronously. The corresponding red arrow refers to the 99.99% confidence interval mark, which is the clock period that, if set, allows not more than 1 in 10000 operations to hit a timing error. The arrow is roughly at the 1.00 mark and thus, in this example, using the 99.99% estimate as guard band is unable to cut out any slack.

On the other hand, assume a design wherein multiple independent operations are executed asynchronously in a sequence i.e. the operations are not separated by clocked elements. The higher *N-level* curves in Fig. 1a correspond to the resultant slack distribution when *N* operations are executed in a multi-cycle internally asynchronous sequence, bounded by synchronous elements.

Via transparent data-flow, the slack accumulates across this sequence and the mean slack estimates are influenced by the number of operations that can be combined together and executed as an asynchronous chain. Note that the greater the lengths of these chains, the higher the average slack per operation - this is explained below. This is because, for independent variation, the estimated mean slack is averaged out over the entire sequence, which would predominantly consist of non-critical operations. The longer the sequence, the more the number of operations that tend to lie closer to the mean value (curve peak increases and width narrows). This would mean that that outliers with a longer critical path can be cushioned by more non-critical operations which consumed less than the high confidence execution time estimate. This allows a lower guard band (i.e. more aggressive clock) for the same confidence interval. In this example, combining 16 operations together (16-level) allows a 20% reduction in the 99.99% guard band estimate.

This statistical representation is a direct interpretation of the central limit theorem (CLT). CLT loosely states that the larger the sample size obtained from a population with a finite level of variance, the more probable it is that the mean across all the samples will be approximately equal to the mean of the population. CLT further states that all of the samples will follow an approximate normal distribution pattern, with all variances being approximately equal to the variance of the population divided by each sample's size.

From this example, it is evident that multi-cycle execution of asynchronous operation sequences provides abundant potential for increased aggressiveness in timing speculation.

3.2 Utilizing transparent pipelines

To exploit the opportunity motivated above, we seek an asynchronous engine design that can integrate seamlessly with standard synchronous pipelined systems/interfaces. Typical circuits with some asynchronous characteristics are: Purely combinational multi-cycle data paths (MDP) [10], Asynchronous Elastic Pipelined (AEP) logic [9] and Transparent pipelines via intelligent latching [5, 4]. MDPs suffer low throughput (or require high replication) and poor flexibility for general purpose programs. AEPs require costly logic for completion detection and handshake mechanisms restricting high frequency implementation, and are harder to interface with synchronous designs.

Thus, we explore transparent latch based pipelines. Latches between FUs are made transparent at appropriate times to allow data to flow through at non-clock boundaries. At other times, the latch is kept opaque preventing dataflow. This allows varied logic delays among operations to be balanced anywhere within the transparent (i.e. asynchronous) execution window.

Fig.1b illustrates the use of a transparent latch (L, in green) between 2 processing elements (PEs). The figure shows the mapping of a function $a*x+b$ onto the 2 PEs, where PE_1 performs a single-cycle multiply and PE_2 performs single-cycle addition. Assume that a, x and b are available at the PE inputs. Note that PE_2 is idling until the multiply operation completes atop PE_1.

The timing diagram (Fig.1c) shows 2 different scenarios - the baseline scenario (refer PE_{1o}, PE_{2o} in figure) which assumes a standard positive edge triggered flip-flop separating the PEs and the transparent scenario (refer PE_{1t}, PE_{2t}) which uses the transparent latch between them. In the former, the flip-flop (F) opens only for a short period of time at the positive clock edge, allowing data to pass through. In the latter, the latch (L) is made transparent for appropriate slack-controlled periods of time. Note: In the figure, a high level (1) for PE_i indicates some computation being performed on that PE.

In the baseline scenario, the multiply operation (on PE_{1o}) is allowed an entire cycle to execute despite the presence of timing slack (shown in red). The addition on PE_{2o} begins only begins after the second positive clock edge. On the other hand, in the transparent design, slack-aware latch control allows L to be open for a period of time which covers the instant at which the multiply completes on PE_{1t}. This allows transparent data flow of $a*x$ into PE_{2t}. Which, in turn, allows PE_{2t} to start real addition computation at the instant of completion of PE_{1t} - thus conserving slack and completing function execution faster than the baseline. Note: the pink dashed line at the end of PE_{1t}'s execution is the error checking period - required due to slack estimation being a speculative mechanism (Sec.4.4).

In summary, we propose a *synchronous slack tracking and opportunistic early clocking* mechanism implemented atop a *transparent pipeline* execution engine. Our proposal *utilizes otherwise idle functional units to capture slack and reduce execution latency*.

4 CONTROL MECHANISM

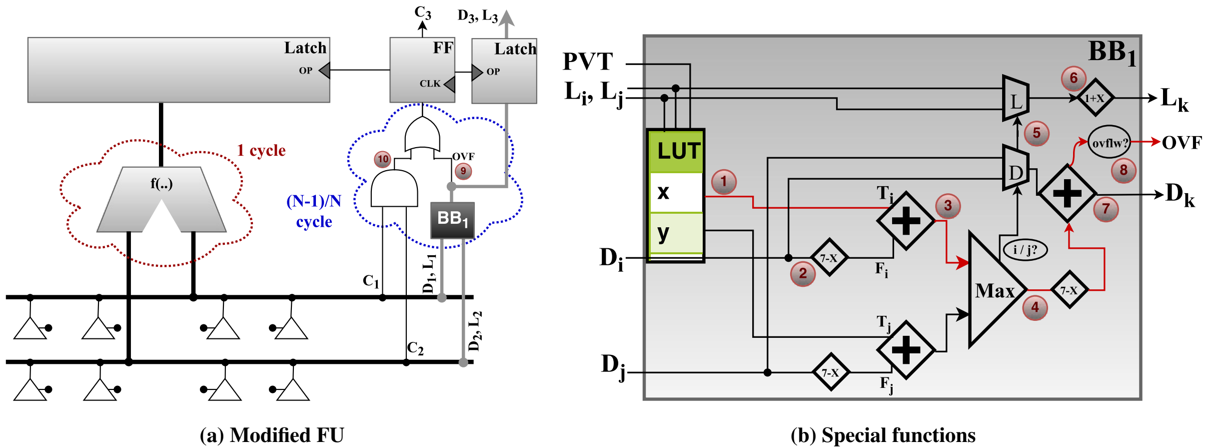
4.1 Slack Estimation

The complexities in accurately estimating the available slack for every operation are tremendously high and therefore exact measurements on a cycle-by-cycle basis are impractical. On the other hand, it is reasonable to make synchronous-domain style slack estimates using the following: ① static design-time information, ② feedback based slack predictor, ③ building a normal distribution model, and ④ extending this to multi-operation sequences using statistical theory. In our analysis, the slack model includes components from the following variations: systematic process, systematic temperature/voltage, random process and data-based. The latter two are modeled as independent/identically distributed (IID) across each operation while the former are correlated across operations.

Tribeca [3] proposed the use of a simple last-value predictor to predict circuit delay behavior under PVT variations, which is then used to tune the processor V/F settings. The predictor chooses the *setting* for the current epoch based on the previous setting and the number of timing violations in the previous epoch. The proposed predictor is within 2% accuracy of oracle prediction. To capture **systematic variations**, we use such a predictor in our design, since our baseline requirement is the same. Such predictors can be appropriately distributed across the chip, so as to pass on localized timing guard band predictions to the proximate compute node(s). Critical Path Monitors [7] could be added to improve slack estimation accuracy further, but we don't explore this possibility. Similar to Tribeca, we make use of a tuning resolution of 10K cycles - but our resolution could be more fine-grained since we do not require costly frequency or voltage tuning.

Fine-grained spatial and temporal **random variation**, along with data-based variations, are not captured from the above. Therefore, to model slack more aggressively (and more accurately) we use statistical modeling aided by static design-time information. We follow prior works [6, 11] that model slack from the probability density function (PDF) of the logic delay as a Gaussian normal distribution with calculated σ and μ values. These distribution parameters are calculated based on estimates of the effects of data inputs as well as PVT variations on logic delay. Sec.6 discusses the estimation of these components in our work.

Statistical slack modeling is especially useful for multi-cycle coalescing based slack estimates, which was discussed earlier in Sec.3.1. The statistical execution time estimate at a particular confidence interval mark (we use 99.99%) is estimated for different length asynchronous operation sequences. These values are written into a look-up table (LUT). The LUT is addressed by the length of the sequence (L) and each entry contains the execution time estimate for the Lth operation in an asynchronous sequence. Based on the position of an operation within its asynchronous sequence, its slack estimate is obtained from the LUT and used appropriately.



(a) Modified FU **(b) Special functions**

Figure 2: ① L_i addresses into LUT to obtain estimation computation times of current operation: T_i, based on i's DFG. Similar for j. ② D_i, the slack accumulated via i's DFG, provides F_i (= 3'b111 - D_i), the completion instant of i within its completion cycle. Similar for j. ③ $F_i + T_i$ is completion time estimate for k based on i. Similarly with j. ④ Conservative estimate for k is assumed from the above, via the $Max()$ operation. ⑤ Depending on i/j being the $Max()$, muxes select constraining producer's D and L. ⑥ L_k is obtained as 1 + constraining L. ⑦ $Max()$ output is converted into slack, and is added to constraining D to create: D_k, the cumulative slack. ⑧ If the cumulative slack overflows, OVF is set. ⑨ OVF set means slack crosses integral boundary and hence early clocking is performed: clocking the operation in the same cycle as the last parent. ⑩ If not, standard clocking is performed, one cycle after completion of the last parent (assuming 1-cycle baseline).

4.2 Slack Accumulation

A computational PE, along with routing logic and slack tracking mechanism is shown in Fig.2.a. Each unit is provided with additional control bits - the executing operation's level in its DFG (L) and the cumulative slack in the operation (op) sequence (D). These bits are propagated along with data flow. Sensitivity analysis for sizing L and D showed 4-bit L (i.e. 16 levels) and 3-bit D (i.e tracking accuracy of 1/8th of cycle) were sufficient for maximizing speedup at minimal design costs. In this scenario the first op after a synchronous boundary with no slack would have $L = 4'b0000$ and $D = 3'b000$. A following dependent op which takes 75% of a cycle to compute, would have $L = 4'b0001$ and $D = 3'b010$.

The hardware performs timing estimation akin to design-time static timing analysis. BB_1 in Fig.2.b shows the dynamic slack estimation mechanism assuming 2 producers (i, j) and one consumer (k). The goal is to estimate D_k and L_k for the consumer (current) op. Fig.2's caption details the design. The slack tracking circuitry is completely in parallel with the data-path and, due to simple logic, has a shorter critical path: thus, no impact on the design's cycle time.

4.3 Early Clocking

Each PE is provided with a completion bit (C) which, when set, indicates that the op it executed has completed. The control mechanism is kept synchronous, so when an op is deemed to be complete at some instant, C is to be set on the subsequent clock cycle boundary.

In this design, let N be the shortest sequence of dependent ops which can accumulate enough slack to shave off one cycle from its execution time (i.e. the synchronous boundary will need to be clocked one cycle early). This would mean that the chain of N ops would complete in $N-1$ clock cycles. This requires N dependent ops' completion bits to be synchronously set in $N-1$ clock cycles. This can be achieved if slack information propagation for this N op sequence can complete in $N-1$ cycles. To achieve this, the slack computation delay per op should be no greater than $(N-1)/N$ cycles (highlighted in Fig.2.a). This is our design's only timing constraint. Further, the propagation of computed slack information should form a (transparent latch based) multi-cycle path. This can be observed in Fig.2.a and Fig.3 wherein slack information propagation bypasses the flip-flop.

The above timing constraint is converted into a bound on the maximum timing slack per op that can be recycled in our design. Let this maximum recyclable slack be s fraction of the clock period - we necessarily forgo any benefit where slack is greater than s. It is intuitive that an N op sequence would complete in $N-1$ clock cycles for the smallest N *iff* each op has maximum slack (= s). Thus $N-1 = N - N*s$ or $N = 1/s$. Substituting this in the earlier result, the slack computation delay per op should thus be less than or equal to $(1-s)$ fraction of a cycle. In other words, the lower the slack computation time, the higher the maximum slack that can be recycled. Note: $s = 0.5$ is the maximum slack we allow in our design and synthesis of slack computation logic easily meets the 0.5 cycle requirement. It should be possible to design for higher or lower slack requirements.

For each computation node, the data latch is made transparent when allocated and is turned opaque in two ways. It could be turned opaque on the next cycle after the last parent op by noting the synchronous completion bits of the parents (i.e. standard synchronous dataflow). It could also be turned opaque on the same cycle as the last completing parent, if the OVF-bit is set. This is because, OVF is set when accumulated slack crosses an integral value, (which causes the slack accumulator to overflow: Fig.2.b). Cumulative slack crossing an integral value means that the current M^{th} op can be clocked in the $M-1^{th}$ cycle (rather than the Mth cycle).

Data flow over 3 single-cycle operations A-C atop this design, is shown in Fig.3: ① Starting at $T = 0$, operation A computes for $0.6ns$. It is synchronously clocked in at the clock boundary i.e. at $T = 1.0ns$ and the slack accumulated is $(1-0.6) = 0.4ns$. ② Via transparent data-flow, operation B can start computing at $T = 0.6ns$

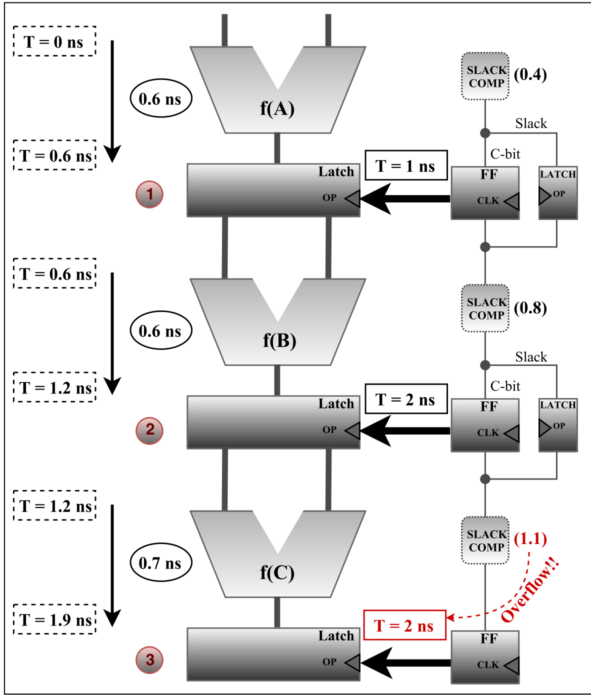


Figure 3: Slack-aware transparent data flow

and computes for another $0.6ns$, ending at $T = 1.2ns$. It is clocked in at $T = 2.0ns$ (one cycle after A) and slack accumulated in total is $(0.4 + 0.4) = 0.8ns$. ③ Similarly, operation C starts compute at $T = 1.2ns$ and completes at $T = 1.9ns$. The total slack accumulated thus far becomes $(0.8 + 0.3) = 1.1ns$ which causes an overflow (by crossing integral boundary). The overflow results in C being clocked in $T = 2.0ns$ i.e. the same clock cycle as B.

4.4 Error Detection and Recovery

We optimize a Razor-like error detection mechanism to suit our requirements. For any single computation, the stable correct output will surely be set within the next s clock cycle fraction. Thus, the inputs to the functional unit are retained for this extra s clock fraction (to avoid short path problems [1]) and error-detection is performed. This detection can complete within the same cycle as the estimated completion or in the next cycle. In the latter case, the inputs are retained at the compute node for the extra cycle.

Error detection is performed via an XOR comparison between the latched *Early* output and the later *Shadow* output. If the PE's output data changes a timing violation has occurred. The latch is then made transparent to capture the correct value and recovery is triggered.

Local data recovery within asynchronous operation sequences involves negligible overheads since transparent data flow is self-correcting across all consumers. Since the latched value in the erroneous FU is corrected after detection, the correct system state can be established by delaying the setting of completion bits (and data latch capture) of younger operations by a single cycle. Further, the slack accumulation is forced to 0 for recovery sequences so that latching correct data is solely controlled by synchronous completion bits. In case recovery spills across synchronous boundaries, overheads are akin to synchronous designs and involves reissue of the synchronous boundary operations and following ones.

Resource	Configuration
CRIB	16 x 4-entry Int. CRIB; 16 x 2-entry FP CRIB
Compute	Int. ALU (1 cycle); FP add (4); FP,mult. (4)
Core Mem.	32 LQ/SQ; 2-w 64K L1I (2); 4-w 32K L1D (2)
Unc. Mem.	8-w 2M L2 (12); Off-chip mem (168)

Table 1: CRIB Specification

Variation	Parameters	Values
PVT sys. (slack %)	Mean, 99.99th	30.5, 19
PVT random	σ/μ (nominal, high)	1.5, 8.5

Table 2: Variation Parameters

5 SPATIAL ARCHITECTURE BASELINE

The design described in Sec.3.2 (Fig.1b) expects a dependent operation (ADD) to be present early (i.e. waiting) at an idle compute unit (PE_2). Such a design can be easily achieved atop a pipelined spatial computing fabric. Spatial fabrics are generally over-provisioned with enough compute resources for high throughput when sufficient application parallelism exists. Under low utilization scenarios these idling/waiting resources can perform slack conservation. Availability of idling/waiting PEs also eases transparent latch control. In traditional transparent pipelines [5, 4], latch management is more complex due to concurrently executing tasks in both the producer stage and the consumer stage, resulting in the need for scheduling bubbles [4], etc. But this is avoided in spatial frameworks which allocate tasks only to free PEs, often far ahead of actual execution.

We implement and evaluate our proposal atop the CRIB spatial architecture [2]. CRIB consists of a matrix of PEs connected by statically routed low latency interconnect. Each PE consists of an ALU and routing logic to read inputs and write outputs. Instructions from the front end are placed into each CRIB entry in program order. Further details are found in [2], while specifications are presented in Table 1. We implement our proposal by provisioning transparent latches between CRIB PEs and adding other components as described in Sec.4. The primary impacting synchronous boundaries are: front-end dispatch and memory operations.

6 METHODOLOGY

Slack modeling: Mean slack estimates for PVT variation and standard deviation for systematic components are obtained from McPAT-PVT [12]. Standard deviation for random variations are obtained for nominal and high estimates from prior work [8, 6]. Numerical values are shown in Table.2. Data slack estimates are obtained via enhancing SoftInj, a software fault injection library that implements the b-HiVE error models [13]. Fig.4 showing statistical slack estimates over an ALU, for datasets corresponding to random inputs, gcc benchmark and the overall SPEC and MiBench benchmark suites. Slack estimates are shown for single operations and asynchronous operation sequences of length 2 and 4. Operations lie closer to mean value for longer sequences, as motivated in Fig.1a

Workloads: We choose SPEC CPU2006 for performance evaluation due to its dynamically varying DFG characteristics, irregular memory access patterns, and hard to predict control flow. We extend

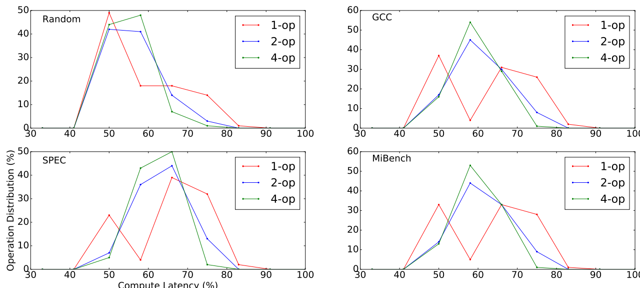


Figure 4: Data Slack Analysis

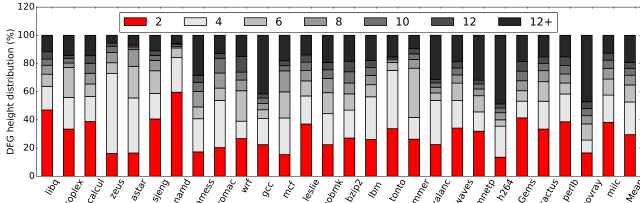


Figure 5: DFG Height Analysis

the Gem5 Simulator to support timing speculation atop CRIB. Results are obtained from Gem5 by running multiple Simpoint slices (each 100M instructions) of the workloads, compiled for ARM ISA.

7 RESULTS

DFG Height Analysis: Fig.5 shows the distribution of DFG heights for SPEC benchmarks. These are the DFGs formed between synchronous boundaries and slack can accumulate across them. For instance, with 25% slack averages, DFGs of height 4 or more accumulate enough slack to clock the end boundaries early. More than 50% of DFGs across most benchmarks allow early clocking.

Performance Speedup: Fig.6 shows speedup from our proposal. Speedup is shown for two random slack distributions: *nominal* and *high* (Table 2). The obtained speedup is broken into two components: benefits obtained from a synchronous *Razor-like* mechanism and atop that, additional benefit obtained from our *Proposal*. The *Razor-like* mechanism is reflective of state-of-art timing speculation where there is no slack accumulation across asynchronous sequences. The *Proposal* components adds additional speedup due to slack accumulation. On average, the total speedups obtained are 18.4% and 10.6% under *nominal* and *high* variations respectively. Within this, *Proposal* provides 32% (*nominal*) and 76% (*high*) higher speedup respectively, over the *Razor-like* implementation (grey vs red portions).

Average speedup is higher under *nominal* random variation (in comparison to *high*) since there is more estimated slack available at the 99.99% confidence requirement. On the other hand, the portion of speedup obtained from *Proposal* is greater under *high* random variation in comparison to *nominal*. This follows from the high confidence requirement, which prevents the *Razor-like* implementation from capturing slack due to slow paths in the slack distribution tail. But the averaging effect of *Proposal*-based speculation moves the slack estimate higher by cushioning latencies of critical instructions with the latencies of more probably non-critical ones.

Design Overheads: Area and energy overhead is calculated by implementing the entire design in RTL and synthesizing with Synopsys Design Compiler at 45nm node. Each PE is provided with slack tracking logic with LUTs and error detection/recovery. The LUTs are designed with 2 read ports and 1 write port, and contain

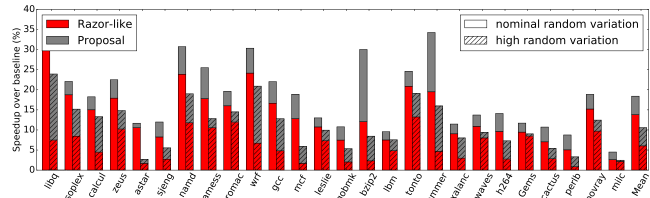


Figure 6: Speedup over baseline

16 entries, each 3-bit wide. The area overhead of slack tracking logic (including LUTs and error logic) is 0.95% atop CRIB.

Energy overhead is 1.12% relative to FU computation energy. This includes 2 LUT reads on each computation, slack accumulation, and error checking logic. LUT writes and error recovery occur roughly once every 10K ops and only marginally add to energy overheads.

8 CONCLUSION

In this work, we proposed a design for aggressive slack recycling by using transparent pipelines. Grouping operations together into an "asynchronous" multi-cycle execution sequence allows timing speculation to cater to the average slack across the group rather than the worst-case individual. This allows more aggressive timing speculation in comparison to completely synchronous mechanisms.

We designed a slack accumulation mechanism and appropriate latch control for early clocking, to achieve slack recycling. Estimated slack is modeled mathematically, with dependence on PVT/data variations along with the height of the multi-cycle DFG, built atop a self-correcting feedback mechanism. The proposal is evaluated on CRIB and shows significant performance speedup under different variation and design constraints.

ACKNOWLEDGEMENTS

The authors would like to thank Keshav Mathur for aiding in preliminary evaluation. This work was supported in part by NSF Award CCF-1615014 and donations from Qualcomm Inc.

REFERENCES

[1] D. Ernst et al. "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation". In: *MICRO 36*. 2003.

[2] E. Gunadi and M. H. Lipasti. "CRIB: Consolidated Rename, Issue, and Bypass". In: *ISCA '11*. 2011.

[3] M. Gupta et al. "Tribeca: Design for PVT Variations with Local Recovery and Fine-grained Adaptation". In: *MICRO*. 2009.

[4] E. Hill and M. Lipasti. "Stall Cycle Redistribution in a Transparent Fetch Pipeline". In: *ISLPED*. 2006.

[5] H. M. Jacobson. "Improved Clock-gating Through Transparent Pipelining". In: *ISLPED '04*. 2004, pp. 26–31.

[6] S. K. Khatamifard et al. "VARIUS-TC: A modular architecture-level model of parametric variation for thin-channel switches". In: *ICCD*. 2016.

[7] C. R. Lefurgy et al. "Active Management of Timing Guardband to Save Energy in POWER7". In: *MICRO-44*. 2011.

[8] J. Liu. "Soft Mousetrap: A Bundled-Data Asynchronous Pipeline Scheme Tolerant to Random Variations at Ultra-Low Supply Voltages". In: *ASYNC*. 2013.

[9] S. M. Nowick. "High-Performance Asynchronous Pipelines: An Overview". In: *IEEE Design Test of Computers* (2011).

[10] J. Sampson et al. "Efficient complex operators for irregular codes". In: *HPCA*. 2011.

[11] S. Sarangi et al. "A Model of Process Variation and Resulting Timing Errors for Microarchitects". In: *IEEE Trans. on Semiconductor Manufacturing* (2008).

[12] A. Tang et al. "Delay and Power Modeling Framework for FinFET Processor Architectures Under PVT Variations". In: *IEEE Trans. on VLSI Systems* (2015).

[13] G. Tziantzioulis et al. "A Bit-level History-based Error Model with Value Correlation for Voltage-scaled Integer and Floating Point Units". In: *DAC*. 2015.